Classification-Based Optimization of Dynamic Dataflow Programs

Hervé Yviquel, Emmanuel Casseau, Matthieu Wipliez, Jérôme Gorin, Mickaël Raulet

Classification-based optimization of dynamic dataflow programs

H. Yviquel[*], M. Wipliez[£], J. Gorin[+], M.Raulet[$], E. Casseau[*]
[*]*IRISA*
Email: herve.yviquel@irisa.fr
Email: Emmanuel.Casseau@irisa.fr
[£]*Synflow SAS*
Email: matthieu.wipliez@synflow.com
[+]*Telecom ParisTech*
Email: jerome.gorin@telecom-paristech.fr
[$]*IETR/INSA*
Email: mraulet@insa-rennes.fr

ABSTRACT

This chapter reviews dataflow programming as a whole and presents a classification-based methodology to bridge the gap between predictable and dynamic dataflow modeling in order to achieve expressiveness of the programming language as well as efficiency of the implementation. We conduct experiments across three MPEG video decoders including one based on the new High Efficiency Video Coding standard. Those dataflow-based video decoders are executed onto two different platforms: a desktop processor and an embedded platform composed of interconnected and tiny Very Long Instruction Word -style processors. We show that the fully automated transformations we present can result in a 80% gain in speed compared to runtime scheduling in the more favorable case.

Keywords: dataflow program, classification, multicore systems, embedded systems, abstract interpretation.

INTRODUCTION

Dataflow programming paradigm was used for years to describe signal processing applications, since the representation of such application in a set of computational units interconnected by communication channel is quite straight forward. Consequently, several kinds of dataflow Models of Computation (MoC), defining the semantic of the programming language, were studied (Johnston, W. M., Paul Hanna, J. R., & Millar, R. J. (2004)). They can be split into two main classes: the static ones allowing a predictable behavior such that scheduling can be done at compile time, and others having a data-dependent behavior need a runtime scheduling (Lee, E. A., & Parks, T. M. (1995)). Most of the studies on dataflow programming focus on the statically schedulable MoC due to efficient synthesis techniques on such models. Unfortunately, they do not take into consideration the flexibility and the expressiveness offered to the programmers by the dynamic dataflow MoC.

The challenge when optimizing the execution of an dynamic dataflow description is then to conserve their strong expressive power while reducing the overhead caused by run-time scheduling. This is the reason why we propose a process that reduces the number of actors that are required to be scheduled at run-time, by clustering network regions that have a locally static

behavior. A locally static region is a set of connected actors in the description that has a firing order we can determine statically.

The chapter is organized as follows. First, the dataflow programming and the context of this study are introduced in Section DataFlow Programming. Then, we explore the work on actor classification in Section Actor Classification. In Section Execution of dynamic dataflow programs present the actor scheduling for dynamic dataflow programs and establish a clustering methodology to optimize it. Finally, we conclude and consider future works.

DATAFLOW PROGRAMMING

The concept of dataflow representation was introduced by Sutherland in 1966 as a visual way to describe a sequence of arithmetic statements (Sutherland, W. R. (1966)). Then, the first dataflow programming language was presented by Dennis in 1974 (Dennis, J. B. (1974)). In this language, a program is modeled as a directed graph where the edges represent the flow of data and the nodes describes control and computation.

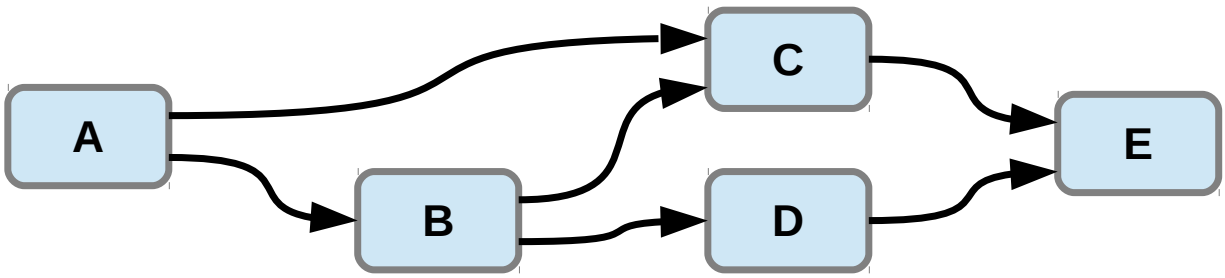

Figure 1: A dataflow graph.

Thus, a dataflow program is defined as a graph composed of a set of computational units interconnected by communication channels. The one presented in Figure 1 contains a network of five components interconnected using communication channels. In the dataflow approach, the communication corresponds to a stream of data composed of a list of tokens.

Properties

During the last twenty years, dataflow programming has been heavily used for the development of signal processing applications due to its consistency with the natural representation of the processing of digital signals. The emergence of parallel programming as a consequence of the frequency wall makes dataflow paradigm an alternative to the imperative paradigm for two reasons:

- the opportunity to use visual programming to describe the interconnection between its components. Such graphical approach is very natural and makes it more understandable by programmers that can focus on how things connect.
- its ability to express concurrency (Johnston, W. M., Paul Hanna, J. R., & Millar, R. J. (2004)) without complex synchronization mechanism. The internal representation of the application is a network of processing blocks that only communicate through the communication channels. Consequently, the blocks are independent and do not produce any side-effect, which remove the potential concurrency issues that arise when asking the programmer to manage manually the synchronization between the parallel computations.

Modularity

The strong separation between the structural and behavioral modeling makes the application description very modular. For instance, a component can easily be replaced by another one while its interfaces (input and output ports) are strictly identical. Moreover, such descriptions are typically hierarchical, in that a component of the graph may represent another graph such as the one in the Figure 2.

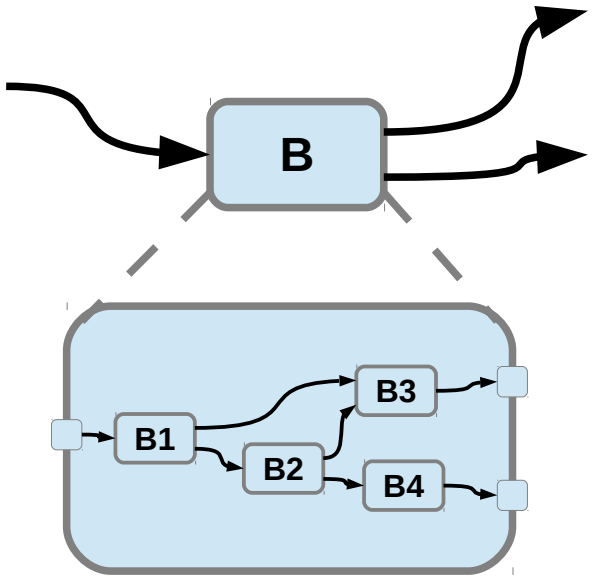

Figure 2: A subnetwork.

Parallelism

A dataflow program states an abundance of parallelism thanks to the explicit exposition of the concurrency. In its structural view, the dataflow model presents three potential degrees of parallelism (task, data and pipeline) that can be applied to different granularities of description.

Additionally, these kinds of parallelism as well as the instruction-level parallelism, i.e. the potential overlap among instructions, can be potentially extracted from the internal algorithm of the components such as any procedural language.

Model of Computation

A Model of Computation (MoC) is an abstract specification of how a computation can progress. A MoC is useful to define the semantics of a programming model, i.e. the type of components it can contain and the way they interact (Savage, J. E. (1998)). Classical examples of MoC are the Turing machine and Lambda calculus models. During the last twenty years, several dataflow MoC were studied due to the attractive use of dataflow programming for the development of signal processing applications.

Existing dataflow MoC can be split into two main classes: the static ones allowing a predictable behavior such that scheduling can be done at compile time, and others having a data-dependent behavior. Most of the studies on dataflow programming focus on the statically schedulable MoC due to efficient synthesis techniques on such models due to their analyzability. Unfortunately, they do not take into consideration the flexibility and the expressiveness offered to the programmers by the dynamic dataflow MoC.

Kahn Process Network

A KPN is represented as a graph $G = (V, E)$ such as V is a set of vertices modeling computational units that are called processes and E is a set of unidirectional edges representing unbounded communication channels based on First In, First Out (FIFO) principles. The behavior of this model of computation can be described using the denotational semantic introduced by Kahn (Kahn, G. (1974)).

A FIFO channel $e \in E$ can be empty, denoted as $\perp$, or can carry a possibly infinite sequence of tokens $X = [x_1, x_2, \dots]$, where each x_i is an atomic data called a token. A sequence X that precedes a sequence Y, e.g. $X = [x_1, x_2]$and $Y = [x_1, x_2, x_3]$, is denoted $X \sqsubseteq Y$. The set of all possible sequences is denoted S, while S^p is the set of p-tuples of sequences on the p FIFO channels of a process. In other words, $[X_1, X_2, \dots, X_p]$ represents the sequence consumed/produced by a process. The length of a sequence is given by $|X|$.

A Kahn process with m inputs and n outputs is a continuous and monotonic function denoted as:

$$F: S^m \rightarrow S^n \quad (1)$$

A process is triggered when S^m appears on it inputs; it is activated iteratively as long as S^m exists. Conversely, the process is suspended when S^m does not exist on its input. In other terms, reading from a FIFO can be blocking for one process until S^m appears again.

The blocking reads insure that every program following this model of concurrency is deterministic. However, it also implies a thread-based implementation of KPN in a sequential environment to backup the current context of the blocked process before executing the next one. Using threads induces inefficiency due to the overhead of the context switching and discard the predictability and determinism of sequential computation (Lee, E. (2006)).

Dataflow Process Network

DPN (Lee, E. A., & Parks, T. M. (1995)), also known as Dynamic dataflow model (DDF), is closely related to KPN. The DPN model is Turing-complete that means it can model any algorithm even non-deterministic ones.

In this model, an application is represented as a graph $G = (V, E)$ within the vertices/processes are called actors. Additionally to the KPN model, it introduces the notion of firing. An actor firing, or action, is an indivisible quantum of computation which corresponds to a mapping function of input tokens to output tokens applied repeatedly and sequentially on one or more data streams. This mapping is composed of three ordered and indivisible steps: data reading, then computational procedure, and finally data writing. These functions are guarded by a set of firing rules R which specifies when an actor can be fired, i.e. the number and the values of tokens that have to be available on the input ports to fire the actor.

More formally, firings can be described using the denotational semantic extended by Dennis (Dennis, J. B. (1974)). Every actor $a \in V$ is associated with its own set of firing function F_a, and firing rules R_a such as:

$$F_a = [f_1, f_2, \ldots, f_M] \quad (2)$$

$$R_a = [\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_N] \quad (3)$$

Within each function $f_i \in F_a$ is associated to a given firing rule $R_i \in R_a$.

A firing rule $\boldsymbol{R}_i$ defines a finite sequence of patterns, one for each input m of the actor such as $\boldsymbol{R}_i = [P_{i,1}, P_{i,2}, \ldots, P_{i,m},]$. A pattern $P_{i,j}$ is an acceptable sequence of tokens in R_i on one input j from the input m of an actor. It is satisfied if and only if $P_{i,j} \sqsubseteq X_j$ where X_j is the sequence of tokens available on the j^{th} FIFO channel. The pattern $P_{i,j} = \perp$ designates any empty list where any available sequence on input j is acceptable. The pattern $P_{i,j} = [*]$ is acceptable for any sequence containing at least one token. The length of a pattern $P_{i,j}$ is denoted $|P_{i,j}|$. We abuse of this notation by using $|R_i|$ to express the consumption rate of the firing rule R_i and $|f_i|$ the production rate of the firing function f_i.

An actor fires when at least one of its firing rules is satisfied. So that, DPN can describe nondeterministic algorithms when several firing rules are satisfied in the same time, which is not possible with the KPN model.

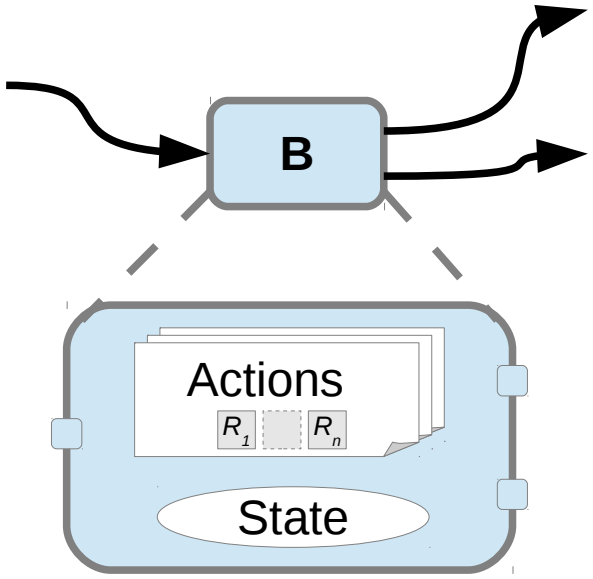

Figure 3: A self-contained actor with its own state, actions and firing rules.

The strong encapsulation of the actors is described by Figure 3 that introduces the internal state of an actor. In fact, such an internal state is just a more convenient representation since it is strictly equivalent to a feedback loop, so it only depends on the ability of the language syntax to describe state variables.

Static dataflow model

The static dataflow model, or synchronous dataflow (SDF), can be seen as a simplification of the DPN model, in which an actor consumes and produces a constant number of tokens at each firing. It may have a single firing rule, which is valid for any sequence S^m of a certain size on its inputs (Lee, E., & Messerschmitt, D. (1987)). In the case where an actor has several firing rules, an actor is SDF if all its firing rules have the same consumption, which mean for $\boldsymbol{R}_a \in$ R and $\forall \boldsymbol{R}_a \in$ R:

$$\boldsymbol{R}_a = \boldsymbol{R}_b \tag{4}$$

All the firing functions of an SDF actor must also produce a fixed number of tokens at each firing, which means for $f_a \in F$ and $\forall f_b \in$ F:

$$|f_a(\mathrm{s})| = |f_b(s)| \tag{5}$$

for any $s \in S^m$ and $s_b \in S^m$

The cyclo-static dataflow MoC (CSDF) (Bilsen, G., Engels, M., Lauwereins, R., & Peperstraete, J. (1996)) extends SDF actors by allowing the number of tokens produced and consumed to vary cyclically. This variation is modeled with a state in the actor, which returns to its initial value after a defined number of firing.

Quasi-static dataflow model

Dataflow modeling is the question of striking the right balance between expressive power and analyzability: On the one hand, synchronous and cyclo-static dataflow limit the algorithms to be modeled as graphs with fixed production and consumption rates for their predictability and their strong properties that allow powerful optimizations to be applied. On the other hand, dynamic dataflow offers a large expressiveness, until Turing-completeness, able to describe complex algorithms with variable and data-dependent communication rate that makes their analyze and optimization ultimately harder.

The needs for a trade-off between expressiveness and predictability has brought the definition of so-called "quasi-static" dataflow models. Quasi-static dataflow differs from dynamic dataflow in that there are techniques that statically schedule as many operations as possible so that only data-dependent operations are scheduled at runtime (Buck, J. (1993), Buck, J., & Lee, E. (1993, Bhattacharya, B., & Bhattacharyya, S. S. (2001)).

Buck's Boolean Dataflow (BDF) model (Buck, J. (1993), Buck, J., & Lee, E. (1993) extends the SDF model with production/consumption rates that depends of a control port with a consumption rate statically fixed at one token by firing. Basically, the rate of a given port p of an actor can be controlled by its associated control port C_p, which means that the actor consumes a token from C_p and the value of this token varies the consumption/production rate of p. The fundamental dynamic actors of the BDF model are the Switch and Select that simply choose one of its two inputs or outputs according to the control token. The BDF model has been proven Turing-complete (Buck, J. (1993)) but it implies a very restrictive coding style that is not very useful for practical cases.

Parameterized dataflow (Bhattacharya, B., & Bhattacharyya, S. S. (2001)) is a higher-level approach to model quasi-static behavior by the extension of existing dataflow model using

parameters modifiable at runtime. For example, Parameterized synchronous dataflow (PSDF) is a generalization of the initial SDF model that allows the expression of quasi-static behavior.

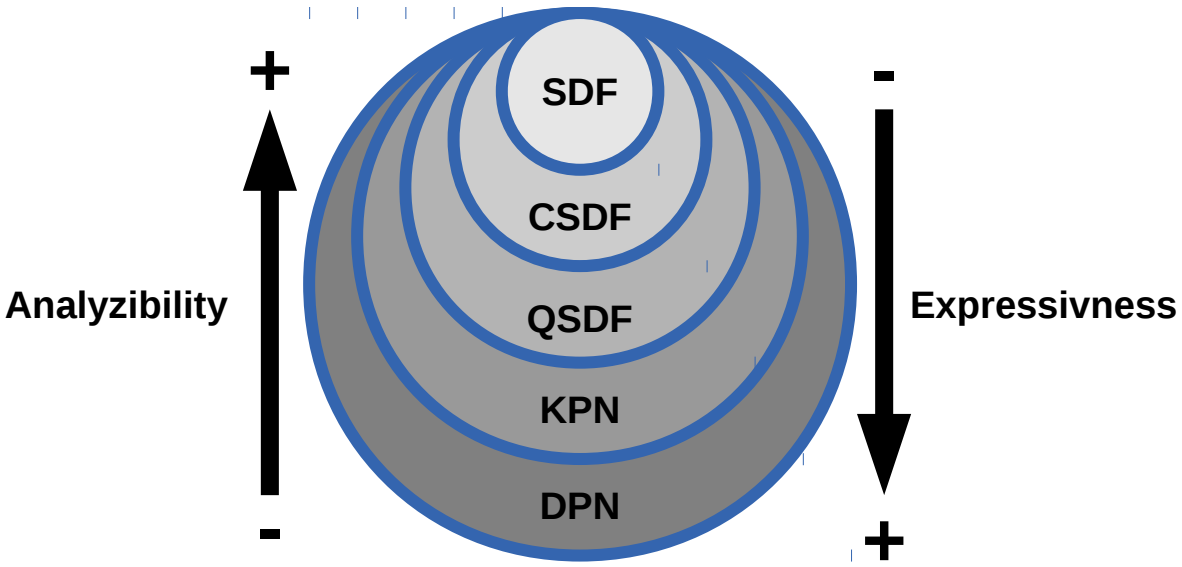

Figure 4: Dataflow Models of Computation.

Dataflow MoCs are defined as subsets of the more general DPN model. The taxonomy shown on Figure 4 reflects the fact that MoCs are progressively restricted from DPN towards SDF with respect to expressiveness, but at the same time they become more amenable to analysis.

Case study: Reconfigurable Video Coding

During the last twenty years, several new programming languages based on dataflow models were proposed by the scientific community to solve a large panel of problems, from parallel programming to signal processing, but only a few have been emerging enough to be involved in the development of real world applications. One of them, the CAL Actor Language (CAL), is particularly interesting due to to the expressiveness offers by its ability to describe data-dependent behavior specified by the DPN MoC (Eker, J. & Janneck, J. (2003).).

The inclusion of a subset of CAL in the MPEG Reconfigurable Video Coding framework enables the development of several video decoders along other applications using dataflow programming. Such a collection of applications offers a great opportunity to study the scheduling of dynamic dataflow programs.

Overview

Reconfigurable Video Coding (RVC) (Mattavelli, M., Amer, I., & Raulet, M. (2010)) is an innovative framework introduced by MPEG to overcome the lack of interoperability between the

several video codecs deployed in the market. The framework is dedicated to the development of video coding tools in a modular and reusable fashion thanks to dataflow programming.

The RVC framework is supported by Orcc[1], an open-source programming toolset based on Model-Driven Engineering technologies, that contains a multi-target compiler as well as an integrated development environment. Orcc is able to translate a unique high-level dataflow program, written in RVC-CAL, into an equivalent description in both hardware and software languages.

Applications

For the experiments presented in this chapter, we use the dataflow descriptions of three MPEG video decoders developed by the RVC group: MPEG-4 Part 2 Simple Profile, MPEG-4 Part 10 Progressive High Profile (also known as AVC or H264) and the new MPEG-H Part 2 (better known as HEVC). Table 1 describes the properties of each description of these well-known decoders: Respectively, the profile of the decoder, the parallelization of the decoding for each component (Luma and Chromas), the number of actors and the number of FIFO channels.

Table 1: Statistics about the RVC-CAL description of several MPEG video decoders.

Decoder	Profile	YUV	#Actors	#FIFOs
MPEG-4 Part 2	SP	yes	41	143
MPEG-4 Part 10	PHP	yes	114	404
MPEG-H Part 2	Main	no	33	81

The description of HEVC, even with a larger complexity, contains a lot less actors than the one of AVC for two reasons: The parallelization of the decoding for each component that induces a duplication of the residual and predication part; The AVC description was written in finer granularity with a lot of control communication.

The RVC-based video decoders are described with a fine granularity (at block level), contrary to the traditional coarse-grain dataflow (at frame level). This fine-grain streaming approach induces a high potential in pipeline parallelism and the use of small communication channels, i.e. their size are defined between 512 and 8192.

[1] The Open RVC-CAL Compiler : A development framework for dataflow programs.

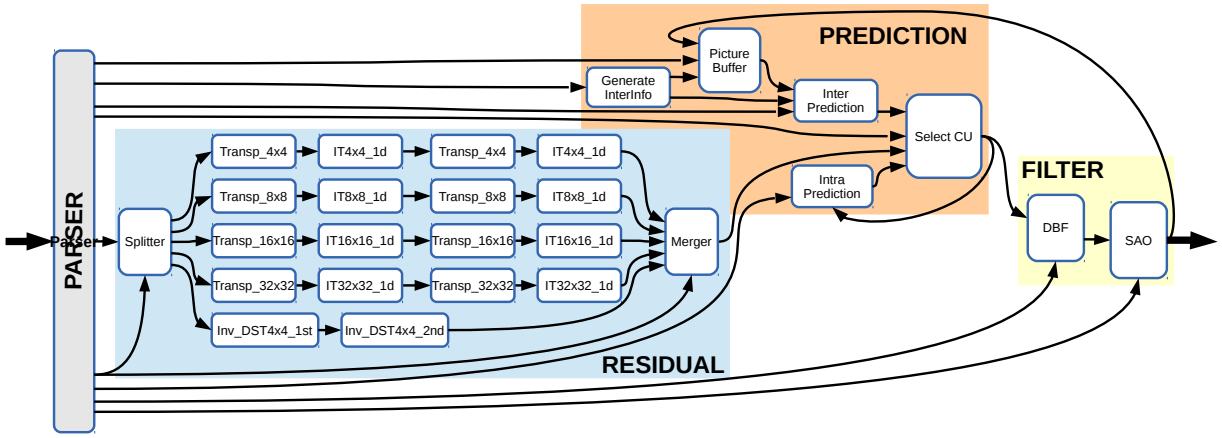

Figure 5: RVC-based description of the MPEG-H Part 2 decoder.

Figure 5 presents the application graph of the MPEG-H Part 2 decoder which could be considered as a good reference of the existing RVC-based video decoders. The decoder is constituted of 4 distinct parts: an actor called Parser which performs the entropy decoding, a subnetwork known as Residual which decodes the prediction residual and another called Prediction which performs the motion compensation as well as the intra prediction and a last one, Filter, containing the deblocking filter and the sample adaptive offset filter. The subnetworks, Residual, Prediction and Filter can be duplicated for each component Y, U and V to increase the data parallelism.

ACTOR CLASSIFICATION

In the simplest case, structural information of an actor is enough to classify it, for instance the rules for an actor to be considered SDF only depend on the input and output patterns of actions. In more complicated cases, it is necessary to gather information from an actual execution of the actor.

Process

The literature introduces several algorithms (Zebelein, C., Falk, J., Haubelt, C., & Teich, J. (2008), Von Platen, C., Eker, J., Nilsson, A., & Arzen, K.-E. (2012), Wipliez, M. (2010), Wipliez, M., & Raulet, M. (2010)) to classify dynamic actors into restricted MoC that can be summed up as follow:

1. **Detection of time-dependent actors:** DPN places no restrictions on the description of actors, and as such it is possible to describe a time-dependent actor in that its behavior depends on the time at which tokens are available. This happens when a given action reads tokens from input ports not read by a higher-priority action, and their firing rules are not mutually exclusive.
2. **Identification of static behavior:** Classification tries to classify each actor within models that are increasingly expressive and complex. The rationale behind this is that the more powerful a model, the more difficult it is to analyze. If an actor cannot be classified as a static actor, the method will try to classify it as cyclo-static, and then as quasi-static. An actor is classified as static iff it conforms to the SDF MoC, which

means that all its actions have the same input and output patterns. A one-action actor is by definition static.

3. **Finding cyclo-static behavior:** An actor has to meet two conditions to be a candidate for cyclo-static classification: it must have a state and there must be a fixed number of data-independent firings that depart from the initial state, modify the state, and return the actor to its original state. Once the actor was identified as a valid cyclo-static candidate, abstract interpretation can be used (see Section Abstract interpretation of Actors) to determine the sequence of actions characterizing its behavior, as well as its production and consumption rates.
4. **Determining quasi-static behavior:** A quasi-static actor is informally described as an actor that may exhibit distinct static behaviors depending on data-dependent conditions. The algorithm is composed of two steps. First, the detection of the input FIFO channels used to control the behavior of the actor and their existing configuration. Then, the identification of static behavior for each configuration using abstract interpretation.
5. **If not** classified in a restricted MoC, the actor is defined as **dynamic**.

Abstract interpretation of Actors

Classifying an actor within a MoC is based on checking that a certain number of MoC-dependent rules hold true for any execution of this actor. Some of these rules are verified solely from the structural information of the actor, for instance the rules for a static actor only depends on the input and output patterns of actions. In more complicated cases, we need to be able to obtain information from an actual execution. The actor must be executed so that the information obtained is valid for any execution of the actor, whatever its environment (the values of the tokens and the manner in which they are available). As a consequence it is not possible to simply execute the actor with a particular environment supplied by the programmer.

To circumvent this problem we use abstract interpretation (Cousot, P., & Cousot, R. (1977)). Abstract interpretation evaluates the computations performed by a program in an abstract universe of objects rather than on concrete objects. The abstract interpretation of an actor has the following properties:

- The set of values that can be assigned to a variable is

$$Values = Z \cup \{true, false\} \cup \{\bot\}$$

The value $\bot$ is used for variables whose value is unknown, e.g. for uninitialized variables.

- The environment is defined as an association of variables and their values:

$$Env: Ident \rightarrow Values$$

Env initially contains the state variables of the actor associated with their initial value if they have one, otherwise with $\bot$.

- When the interpreter enters an action, the environment is augmented with bindings between the name of the tokens in the input pattern and $\bot$. In other words, a token read has an unknown value by default.

The abstract interpreter interprets an actor by firing it repeatedly until either one of the conditions is met:

1. The interpreter is told to stop because analysis is complete as determined by the classification algorithm.
2. The interpreter cannot compute if an action may be fired because this information depends on a variable whose value is ⊥.

To fire the actor, the interpreter starts by selecting one fireable action, which is an action that meets all its firing rules. As far as the quantity of tokens is concerned, the abstract interpretation models infinite FIFOs channel, which means an action always has enough tokens to fire. Other differences between concrete interpretation of an actor, and its abstract interpretation include the following. Any expression that references a variable v where $Env(v) = \perp$ has the value ⊥. Conditional statements and loops that test an expression whose value is ⊥ are not executed. However, guards evaluated as cause the abstract interpreter to stop as per condition 2.

Evaluation

We have tested the classification on the actors included in the RVC-based descriptions of the three tested video decoders. Table 2 shows the number of actors classified as static, cyclo-static, quasi-static, dynamic and time-dependent.

Table 2: Classification results for MPEG-4 Part 2 SP (a), MPEG-4 Part 10 PHP (b), MPEG-H Part 2 (c).

(a)		(b)		(c)		
#	%	#	%	#	%	
6	15	42	37	19	58	Static
8	20	5	4	1	3	Cyclo-static
3	1	5	4	0	0	Quasi-static
23	56	56	49	10	30	Dynamic
1	0.5	6	5	3	9	Time-dependent

The results show that only a small percentage of the whole set of actors are time-dependent and quasi-static, since a vast majority is classified as static and dynamic. Most of the static actors correspond to ~~the~~ algorithms that decode the residual picture. Only a few actors was detected with quasi-static behavior, this is a direct consequence of the classification algorithm that only handle simple case due to the difficulty to analyze it in all cases.

The granularity of the actors are very different: the static and cyclo-static actors are generally less than 50 lines of code, and most of dynamic actors are a lot larger, frequently between 500 and 1000 lines. This makes it unlikely to find actors that behave according to more restricted MoCs, and leads us to believe that the classification method will yield the best results on applications described with fine-grain, small actors.

EXECUTION OF DYNAMIC DATAFLOW PROGRAMS

The strong encapsulation of components in the dataflow execution model offers an explicit modeling of the concurrency within an application. A natural approach for handling concurrent execution on a sequential environment is the use of threads. However, thread-based implementations, on top of introducing non-determinism (Lee, E. (2006)), can lead to a large overhead when a large number of components are executed to the same processing unit (Carlsson, A., Eker J., Olsson, T. & Von Platen, C. (2010)).

Instead of relying on threads managed by the operating system kernel, the DPN model allows a continuous execution of the operations of a graph thanks to a user-level scheduler (Lee, E. A., & Parks, T. M. (1995)). This scheduler can sequentially test the firing rules from several actors, and fire an actor if a firing rule is valid. An efficient scheduling for dataflow programs consists in finding a, pre-defined or not, order of actor firings throughout the execution process capable of maximizing the use of all the processing units in one platform. Since actors in a DPN may have data-dependent behaviors, and the data are unknown in the system, determining an optimal schedule of a program is not possible at compile-time (equivalent to the halting problem (Parks, T. M. (1995))), i.e. the scheduling can be only done in the general case at run time.

The section describes several scheduling strategies designed to execute a DPN-based application on a single-core architecture, which handles the execution of only one actor at a given time. The strategies are then evaluated on two different platforms in order to be compared.

Round-robin

It is a simple scheduling strategy that continuously goes over the list of actors: The scheduler evaluates the firing rules of an actor, fires the actor if a rule is met and continues to evaluate the same actor until no firing rules are met then switches to the next actor. This scheduling policy guarantees to each actor an equal chance of being executed, and avoids deadlock and starvation. Contrary to classical round-robin scheduling, there is no notion of time slice: an actor is executed until it cannot fire anymore in order to minimize the number of actor switching and consequently the scheduling overhead. The reason of this actor switching is that in practice the FIFO channels will be finally full or empty because of their bounded sizes.

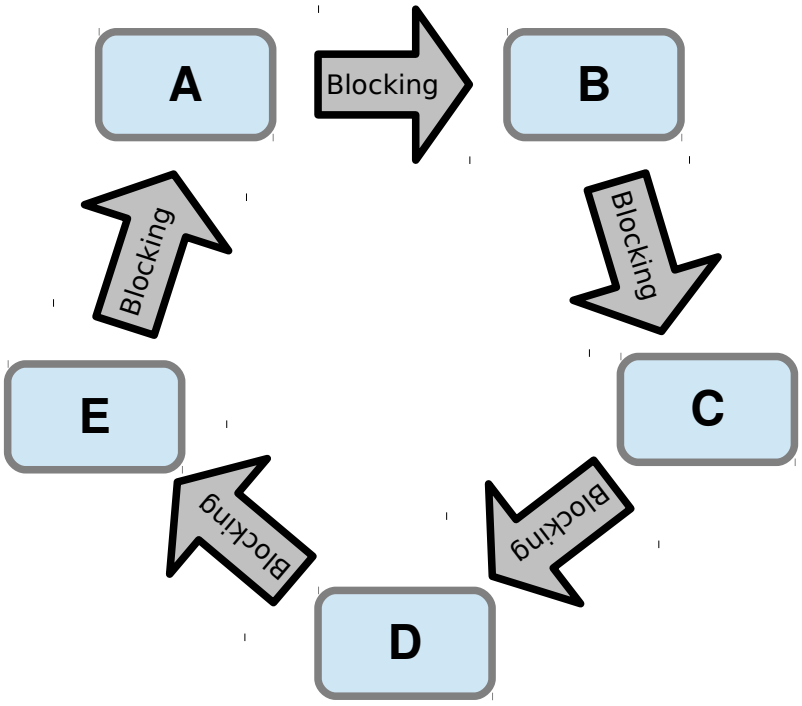

Figure 6: Example of round-robin scheduling with five actors.

Figure 6 shows an application of this round-robin scheduling on the example of dataflow graph presented in Figure 1. The scheduler executes the actors in a circular order i.e. the five actors A, B, C, D and E are successively executed then the scheduler starts again from A and so on.

Data-driven / demand-driven

This strategy is a more advanced runtime scheduling strategy. Indeed, the round-robin strategy schedules actors unconditionally i.e. the firing rules of an actor could be checked even if they are all invalid. In this case, the firing rules of the actor will be checked, but no computation will be performed, that is called a miss. As a result, the round-robin strategy becomes inefficient with complex applications containing hundred of actors and a lot of control communications.

Data-driven / demand-driven scheduling (Yviquel, H., Casseau, E. Wipliez, M. & Raulet, M. (2011)) strategy is based on the well-known data driven and demand driven principles (Parks, T. M. (1995)). On the one hand, data-driven policy executes an actor when its input data have to be consumed to unblock the execution of the precedent actor. On the other hand, demand-driven executes an actor when its output is needed by one of its successor actor.

Two types of events can cause the blocking of an actor execution, each one is implying a different scheduling decision:

- When an actor is blocked because an input communication channel is empty, demand-driven policy is applied and ask the scheduler to execute the predecessor of this channel.

- When an actor is blocked because an output communication channel is full, data-driven policy is applied and ask the scheduler to execute the successor of this channel.

Contrary to the round-robin algorithm, a dynamic list of next schedulable actors is needed. The behavior of this schedulable list is illustrated with Figure 7. When an actor is blocked during its execution, the empty or full FIFOs are identified and their associate predecessors or successors are added to the schedulable list. The actor to be executed next corresponds to the next entry in the schedulable list.

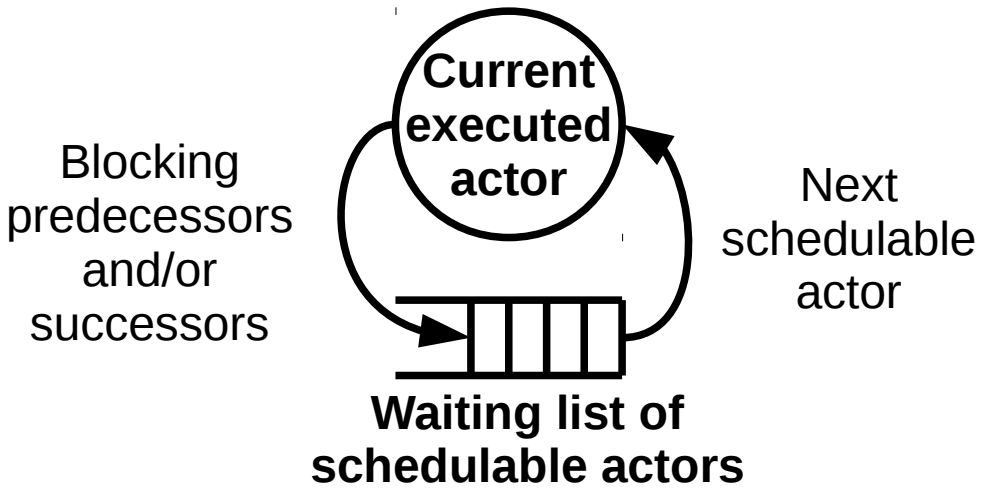

Figure 7: Behavior of the dynamic list of next schedulable actor used by data-driven / demand-driven scheduling.

Actor machine

Round-robin, as well as data-driven/demand-driven, strictly respects the execution model of DPN MoC defined by Lee and Parks (Lee, E. A., & Parks, T. M. (1995)). The execution of an actor is modeled by the repeated evaluation of the firing rules that are, in case of a success, followed by the firing of the associated firing function, also known as action. The firing rules can evaluate two kinds of condition, the amount of tokens available and the values of this tokens, that ultimately leads to execution of a large number of tests. A different approach, introduced by Janneck et Cedersjö, tries to reduce the number of test performed during the evaluation of firing rules using a new execution model, called actor machine (Cedersjo, G. & Janneck, J. W. (2012), Janneck, J. W. (2011)), that also considers the evaluation results of previous firing rules.

Actor machine deals with the memorization of the test results involved in the validation of previous firing rules to limit their reproduction. For instance, let two firing rules R_i and R_j tested successively such as $R_i = [P_{i,1}, P_{i,2}]$ and $R_j = [P_{j,1}, P_{j,2}]$ with$P_{i,1} = P_{j,1} = [*,*]$; if R_i is evaluated false such as $R_i = [true, false]$ then $P_{j,1}$ could be already known valid during the evaluation of R_j and the evaluation of $P_{j,2}$ should be sufficient. To do so, the evaluations of previous patterns are preserved using an automaton mechanism. Several connected actor machines can also be composed in order to increase the potential possible reduction (Janneck, J. W. (2011)).

On the one side the scheduling of an actor machine could be more efficient compared to the traditional firing model thanks to the reduction of the number of tests performed; But on the other the translation to the actor machine execution model induces an explosion of the number of states in the scheduling algorithm due to the need of memorization. Moreover, a circular buffer implementation of the communication channel can permit an equivalent test reduction by mean of compiler optimization. Indeed common sub-expression elimination can search for identical patterns in firing rules evaluated successively, and replaces them with a single variable holding the result of their evaluation.

Quasi-static scheduling

In the previous sections, we have stated that one essential benefit of the DPN model lies in its strong expressive power, so as to simplify algorithm implementation for programmers. This expressive power includes: the ability to describe data-dependent computations through token production/consumption, where production/consumption may vary according to values of tokens; the ability to express non-determinism, which can be used to construct actors that respond to unpredictable sequences of tokens; and, the ability to produce time-dependent behaviors that rely on the time at which tokens are available on the input of an instance.

However, when dealing with the scalability of this model, we have stated that this strong expressive power incurs a cost on the efficiency of its implementation, as several operations may be scheduled at run-time on a single processing unit. The overhead caused by a scheduling strategy, along with its variable chance of success between test/validation of a firing rule for each operation, can create a succession of synchronization issues between the firing of actors in a description. This issue can ultimately lead to inefficient implementation of dataflow programs or to unsteady performance on their executions. The granularity of an application, i.e. the number of actors to schedule in the description, becomes an important factor that can prevent synchronization issue of instances.

The challenge when optimizing the execution of a dataflow description is then to conserve the strong expressive power of DPN while reducing the overhead caused by its required run-time scheduling. Quasi-static scheduling intends to make scheduling decisions as much as possible at compile-time by determining all static behavior and keeping only the necessary decision for run-time. The literature has introduced a large panel of methodologies to perform quasi-static scheduling of dynamic dataflow programs in different manner (Gorin, J., Wipliez, M., Prêteux, F. & Raulet, M. (2011), Gu, R., Janneck, J., Bhattacharyya, S., Raulet, M., Wipliez, M., & Plishker, W. (2009), Ersfolk, J., Roquier, G., Jokhio, F., Lilius, J. & Mattavelli, M. (2011), Ersfolk, J., Roquier, G., Lilius, J., & Mattavelli, M. (2012), Boutellier, J., Raulet, M. & Silven, O. (2013), Boutellier, J., Lucarz, C., Lafond, S., Gomez, V., & Mattavelli, M. (2009), Boutellier, J., Silven, O. & Raulet, M. (2011)).

Some of them try to prune all unreachable execution paths to remove all unnecessary tests using code instrumentation (Boutellier, J., Raulet, M. & Silven, O. (2013), Boutellier, J., Lucarz, C., Lafond, S., Gomez, V., & Mattavelli, M. (2009), Boutellier, J., Silven, O. & Raulet, M. (2011)) or model checking (Ersfolk, J., Roquier, G., Jokhio, F., Lilius, J. & Mattavelli, M. (2011), Ersfolk, J., Roquier, G., Lilius, J., & Mattavelli, M. (2012)) to determined the possible executions. However, both of them are limited by their need of input data to perform their analysis. Such a requirement prevents the full support of all possible execution paths.

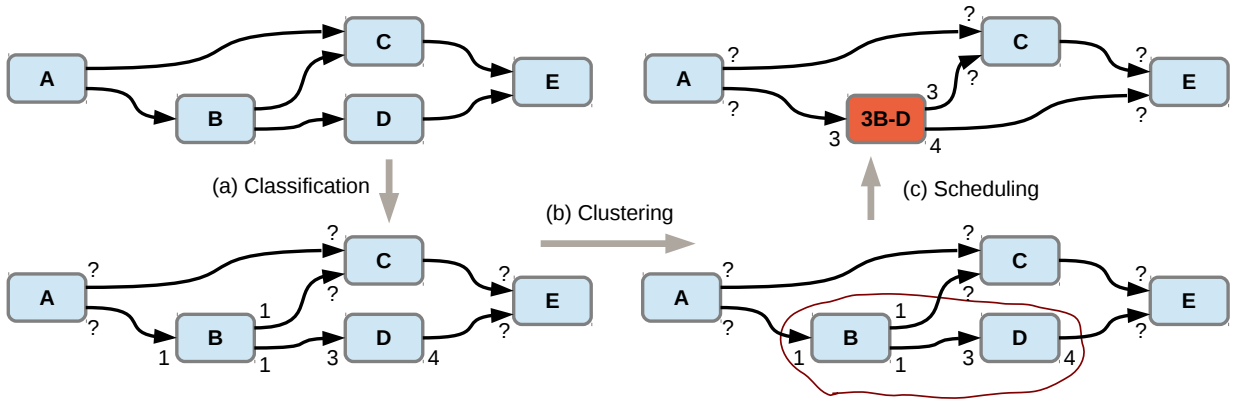

Figure 8: Quasi-static scheduling using actor clustering.

Another approach, based on the classification results, try to reduce the number of actors that are required to be scheduled at run-time, by clustering network regions that have a locally static behavior (Gorin, J., Wipliez, M., Prêteux, F. & Raulet, M. (2011), Gu, R., Janneck, J., Bhattacharyya, S., Raulet, M., Wipliez, M., & Plishker, W. (2009)). We mean by one locally static region a set of connected actors in the description that has a firing order we can determine statically, regardless the data stored in the FIFO channels of the description. The actor clustering approach is based on three existing algorithms that are applied sequentially as follow:

1. The actors with predictable behaviors present in the dataflow description are detect using actor classification as described in Section Actor Classification,
2. Predictable actors connected amongst themselves are clustered into a single node, the composite node to obtain a valid sequence of firing in it that can be determined at compile-time. As such, an essential condition to set a composite node is to determine whether such a sequence of firing is possible, the composition theorem described by Pino (Pino, J. L. & Lee, E. (1995)). The resulting cluster becomes a composite node in the graph of the dataflow description,
3. Actors grouped in a composite node are scheduled the Single-Appearance Scheduling (SAS) (Oh, H., Dutt, N. & Ha, S.. (2005)), the optimum static scheduling strategy for code minimization where all repetitions of a same actor can be found side by side. The other remaining actors, along with the resulting composite nodes, are scheduled at run-time.

The methodology is illustrated on the Figure 8 on a dataflow example containing 5 actors. Each actor is firstly classified to determine, if possible, its production/consumption rates in order to detect the existing static region that can finally be scheduled.

Evaluation

Quasi-static scheduling based on local regions clustering has been evaluated on the tested video decoders. Table 3 presents the number of clusters found in each description, as well as the number of actors and FIFO channels affected by the clustering.

Table 3: Clustering results.

	Clusters	Actors	FIFOs

	#	#	%	#	%
MPEG-4 Part 2	3	6	15	3	2
MPEG-4 Part 10	8	30	26	22	5
MPEG-H Part 2	5	18	55	13	16

As a more detailed example, we focus on the inverse transformation composing the description of the MPEG-H Part 2 decoder, see Figure 5. This Inverse Transformation is decomposed in five parallel paths. The first paths are dedicated to the prediction residual decoding of each existing transform units (TUs) sizes, respectively 4x4, 8x8, 16x16 and 32x32. The last one is used to compute the inverse discrete sine transform (DST) for 4x4 luma transform blocks that belong to an intra coded region.

All actors of the Inverse Transformation are classified static except the splitter and the merger. A clustering node is build for each paths with the schedule presented in Figure 9

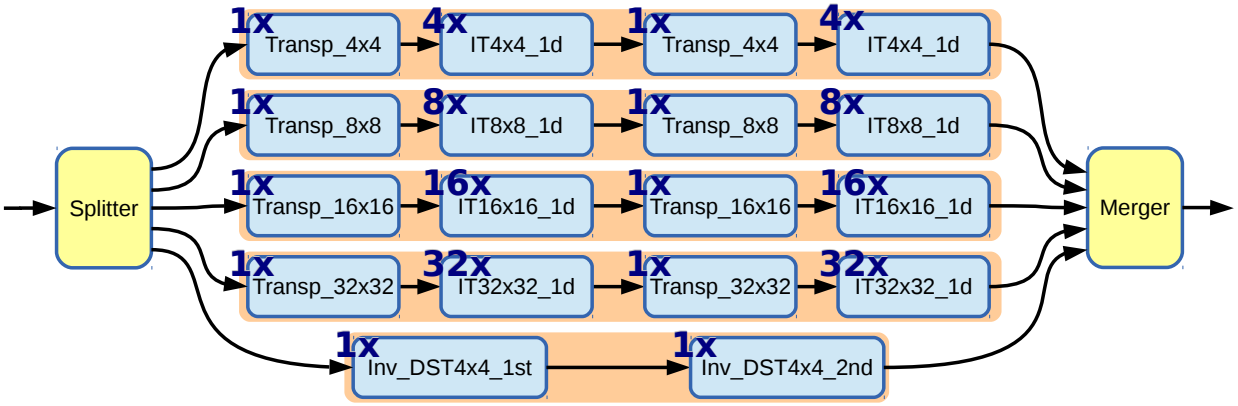

Figure 9: Quasi-static scheduling of the Inverse Transformation of MPEG-H Part 2.

The experiments have been made from the following 720P sequences containing I/P/B frames:

- MPEG-4 Part 2: Old town cross - 25fps, 6Mbps
- MPEG-4 Part 10: A Place at the Table - 25fps, 6Mbps
- MPEG-H Part 2: Four People - 60fps, 1Mbps

The Table 4 summarizes the results of the sequential execution of the tested application with different scheduling approaches, that are the round-robin and data-driven/demand-driven strategies with or without the clustering of locally static regions. The number of switches, firings and misses is detailed, additionally to the percentage of misses according to the total number of switches. A switch corresponds the execution of the next schedulable actor that occurs when the current actor is not fireable anymore due to the empty/full state of its communication channels; A firing corresponds to the execution of an action when all its firing rules are valid; And a miss is a switch that do not result to a firing.

Table 4: Comparison of mono-processor scheduling strategies. The number of switches, firings and misses are expressed in 10^3.

(a) MPEG-4 Part 2 SP

Strategy	Switch	Firing	Miss	%
Robin	4306	2812677	801	19
Robin+Cluster	N/A	N/A	N/A	N/A
Driven	2420	2812654	252	10
Driven+Cluster	N/A	N/A	N/A	N/A

(b) MPEG-4 Part 10 PHP

Strategy	Switch	Firing	Miss	%
Robin	522205	12526470	447065	86
Robin+Cluster	N/A	N/A	N/A	N/A
Driven	52096	12539650	25633	49
Driven+Cluster	50348	11824252	26369	52

(c) MPEG-H Part 2 Main

Strategy	Switch	Firing	Miss	%
Robin	16389	232752	12855	78
Robin+Cluster	9929	226807	147	1
Driven	2808	233057	147	5
Driven+Cluster	2647	227228	91	3

From desktop processors

First of all, the scheduling strategies are evaluated on a desktop processor, an Intel Core i7 M640 clocked at 2.80GHz using the ANSI C backend of Orcc.

Table 5: Comparison of the frame-rate, in FPS, of the tested video decoders on 720p sequences using different scheduling approaches.

Strategy	Robin		Driven	
Clustering	No	Yes	No	Yes
MPEG-4 Part 2	22.5	23	22.5	22.7
MPEG-4 Part 10	4.7	4.7	5.6	7.8
MPEG-H Part 2	14.7	14.6	15.3	15.1

Table 5 presents the frame-rate of the tested decoder using the different scheduling strategies. The results show that even an impressive reduction of misses in the scheduling does not induce an improvement of the global performance of the application in term of frame-rate.

Past experimentations (Wipliez, M., & Raulet, M. (2010), Gorin, J., Wipliez, M., Prêteux, F. & Raulet, M. (2011), Gu, R., Janneck, J., Bhattacharyya, S., Raulet, M., Wipliez, M., & Plishker, W. (2009)) have presented interesting improvements of the performance of dynamic dataflow application using static scheduling of locally predictable regions. This difference can be explained by a more efficient implementation of the FIFO mechanism in the compiler. In the dataflow approach, the communication is one of the well-known bottleneck of the performance and one of the interest of actor clustering is the replacement of the channel by temporary variables. Consequently, the more efficient the implementation of those communication channel is, the less impressive the performance improvement induced by actor clustering is.

Such results can also be explained by the efficiency of desktop processors to handle the heavy control-oriented algorithm induce by dynamic dataflow models. In fact, contemporary processors use a large number of techniques to handle any unpredictability such as out-of-order execution and branch predicator.

Towards embedded platforms

The embedded multi-core platform is based on a VLIW-style architecture known as Transport-Trigger Architecture (TTA) (Corporaal, H. (1997)). TTA processors resemble VLIW processors in the sense that they fetch and execute multiple instructions each clock cycle. TTA processors do not support advanced hardware technology such as branch predicator or out-of-order execution. The architectural simplicity, joined with the extensive capacity of computation, makes it an interesting target for embedded platform. However, these properties make it very sensitive in term of performance to the control present in the application.

Table 6: Scheduling results for xIT of HEVC

Region	Tokens	Round-robin	Clustering	Acc

IT4x4	13696	385876	272982	1.41
IT8x8	38848	986408	664708	1.48
IT16x16	42752	2917961	1557331	1.87
IT32x32	19424	1747047	1567697	1.11

Run-time and quasi-static scheduling are both experimented on the inverse transformation, described in Figure 9, to compare their efficiency on a such platform. In one hand, the four actors composing the path are mapped to the same processor and scheduled dynamically with the round-robin policy. On the other hand, the four actors are clustered and scheduled using SAS in a composite node at compile-time then the resulting node is mapped to an equivalent processor. Table 6 presents the important acceleration rates obtained with the quasi-static scheduling for each TUs sizes. Such speed-ups show that the TTA processor benefits a lot more from to the compile-time predictability, offering more potential instruction-level parallelism, than the desktop processors.

CONCLUSION

The emergence of massively parallel architecture, along with the necessity of new parallel programming models, has revived the interest on dataflow programming due to its ability to express concurrency. As discussed throughout this chapter, although dynamic dataflow programming can be considered a flexible approach for the development of scalable application, there are still some open problems in concern of their execution. Since most of the literature stays focus on the study of predictable dataflow models due to their analyzability, the detection of static behavior in dynamic description aims to bridge the gap between both worlds.

As much of the experiments of this chapter demonstrate, the interest of quasi-static scheduling of dynamic dataflow programs is strongly dependent of the ability of the targeted platform to handle unpredictable behavior; a key feature therefore, is the ability of the analysis to determine the MoC of a given actor. Future research interests will include the improvement of the detection and scheduling of quasi-static MoCs, as well as, the study of such techniques in the context of a multi-core platform.

REFERENCES

Bhattacharya, B., & Bhattacharyya, S. S. (2001). Parameterized Dataflow Modeling for DSP Systems. *IEEE Transactions on Signal Processing*, 49, 2408–2421.

Bilsen, G., Engels, M., Lauwereins, R., & Peperstraete, J. (1996). Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2), 397–408.

Boutellier, J., Lucarz, C., Lafond, S., Gomez, V., & Mattavelli, M. (2009). Quasi-static scheduling of CAL actor networks for Reconfigurable Video Coding. *Journal of Signal Processing Systems*, 1–12.

Boutellier, J., Raulet, M. & Silven, O. (2013). Automatic Hierarchical Discovery of Quasi-Static Schedules of RVC- CAL Dataflow Programs. In Journal of Signal Processing Systems, 71(1):35 – 40.

Boutellier, J., Silven, O. & Raulet, M. (2011). Scheduling of CAL actor networks based on dynamic code analysis. *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*.

Buck, J. (1993). Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis, University of California, Berkeley.

Buck, J., & Lee, E. (1993). Scheduling dynamic dataflow graphs with bounded memory using the token flow model. *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 429-432.

Carlsson, A., Eker J., Olsson, T. & Von Platen, C. (2010). Scalable parallelism using dataflow programming. Ericsson Review, 2(1):16 – 21.

Cedersjo, G. & Janneck, J. W. (2012). Toward Efficient Execution of Dataflow Actors. In Signals, Systems and Computers (ASILOMAR) pages 1465 – 1469.

Corporaal, H. (1997). Microprocessor Architectures: from VLIW to TTA . John Wiley & Sons, Chichester, UK.

Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Sigact-Sigplan Symposium on Principles Of Programming Languages* (pp. 238–252).

Dennis, J. B. (1974). First Version of a Data Flow Procedure Language. In Programming Symposium, pages 362 – 376. Springer Berlin Heidelberg.

Eker, J. & Janneck, J. (2003). CAL Language Report. *Technical Report ERL Technical Memo UCB/ERL M03/48*, University of California at Berkeley.

Ersfolk, J., Roquier, G., Jokhio, F., Lilius, J. & Mattavelli, M. (2011). Scheduling of dynamic dataflow programs with model checking. In Systems (SiPS), pages 37 – 42.

Ersfolk, J., Roquier, G., Lilius, J., & Mattavelli, M. (2012). Scheduling of dynamic dataflow programs based on state space analysis. In IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP-12. pages 1661 – 1664.

Gorin, J., Wipliez, M., Prêteux, F. & Raulet, M. (2011). LLVM-based and scalable MPEG-RVC decoder. *Journal of Real-Time Image Processing, Springer*.

Gu, R., Janneck, J., Bhattacharyya, S., Raulet, M., Wipliez, M., & Plishker, W. (2009). Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis. *IEEE Transactions on Circuits and Systems for Video Technology*.

Janneck, J. W. (2011). A machine model for dataflow actors and its applications. In Signals, Systems and Computers (ASILOMAR) pages 756 – 760, USA.

Johnston, W. M., Paul Hanna, J. R., & Millar, R. J. (2004). Advances in dataflow programming languages. ACM Computing Surveys , 36(1):1 – 34.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Proceedings of IFIP'74* (p. 471-475).

Lee, E. (2006). The problem with threads. Computer, 39(5):33 – 42.

Lee, E., & Messerschmitt, D. (1987). Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1235–1245.

Lee, E. A., & Parks, T. M. (1995). Dataflow Process Networks. *Proceedings of the IEEE*, 83(5), 773–801.

Mattavelli, M., Amer, I., & Raulet, M. (2010). The Reconfigurable Video Coding Standard [Standards in a Nutshell]. IEEE *Signal Processing Magazine*, 27(3), 159 -167.

Oh, H., Dutt, N. & Ha, S.. (2005). Single appearance schedule with dynamic loop count for minimum data buffer from synchronous dataflow graphs. In Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems - CASES '05, pages 157 – 165, USA. ACM.

Parks, T. M. (1995). Bounded Scheduling of Process Networks. *Doctoral dissertation*, Berkeley, CA, USA.

Pino, J. L. & Lee, E. (1995). Hierarchical static scheduling of dataflow graphs onto multiple processors. In International Conference on Acoustics, Speech, and Signal Processing. ICASSP-95, pages 2643 – 2646.

Savage, J. E. (1998). Models of computation: exploring the power of computing. Addison-Wesley Pub.

Sutherland, W. R. (1966). The on-line graphical specification of computer procedures. PhD thesis, MIT.

Von Platen, C., Eker, J., Nilsson, A., & Arzen, K.-E. (2012). Static Analysis and Transformation of Dataflow Multimedia Applications. Technical report, Lund University.

Yviquel, H., Casseau, E. Wipliez, M. & Raulet, M. (2011). Efficient multicore scheduling of dataflow process networks. In IEEE Workshop on Signal Processing Systems (SiPS), pages 198 – 203, USA.

Wipliez, M. (2010). *Compilation Infrastructure for Dataflow Programs*, Ph.D. dissertation, National Institute of Applied Sciences (INSA).

Wipliez, M., & Raulet, M. (2010). Classification and Transformation of Dynamic Dataflow Programs. In *Design and Architectures for Signal and Image Processing (DASIP)*.

Zebelein, C., Falk, J., Haubelt, C., & Teich, J. (2008). Classification of General Data Flow Actors into Known Models of Computation. Proc. *MEMOCODE*, Anaheim, CA, USA, 119–128.